VB2

AN ARCHITECTURE FOR INTERACTION IN SYNTHETIC WORLDS

Enrico Gobbetti, Jean-Francis Balaguer

Daniel Thalmann
Computer Graphics Laboratory
Swiss Federal Institute of Technology
CH-1015 LAUSANNE, Switzerland

{gobbettilbalaguer}@di.epfl.ch

ABSTRACT

This paper describes the *VB2* architecture for the construction of three-dimensional interactive applications. The system's state and behavior are uniformly represented as a network of interrelated objects. Dynamic components are modeled by *active variables*, while multi-way relations are modeled by *hierarchical constraints*. *Daemons* are used to sequence between system states in reaction to changes in variable values The constraint network is efficiently maintained by an incremental constraint solver based on an enhancement of *SkyBlue*. Multiple devices are used to interact with the synthetic world through the use of various interaction paradigms, including immersive environments with visual and audio feedback. Interaction techniques range from *direct manipulation*, to *gestural input* and *three-dimensional virtual tools*. Adaptive pattern recognition is used to increase input device expressiveness by enhancing sensor data with classification information. Virtual tools, which are encapsulations of visual appearance and behavior, present a selective view of manipulated models' information and offer an interaction metaphor to control it. Since virtual tools are first class objects, they can be assembled into more complex tools, much in the same way that simple tools are built on top of a modeling hierarchy. The architecture is currently being used to build a virtual reality animation system.

KEYWORDS

User Interface Design, 3D Interaction, 3D Virtual Tools, Gestural Input, Virtual Reality, Object-Oriented Graphics, Hierarchical Constraints.

1. INTRODUCTION

The latest high-speed graphics workstations and devices make it possible to create applications in which the user directly manipulates aspects of three-dimensional synthetic worlds, ideally without feeling the mediation of a computer. However, most of today's user interfaces for 3D graphics systems still predominantly use 2D widgets, direct interaction with the 3D world being generally limited to interactive viewing, selection, positioning and manipulation of points on paths or patches.

The difficulties associated with achieving the key goal of immersion has led the research in virtual environments to concentrate far more on the development of new input and display devices than on higher-level techniques for 3D interaction. It is not until recently that interaction with synthetic worlds has tried to go beyond straightforward interpretation of physical device data [27][2]. AT&T's embryonic CAD modeler [39] is an example of a system showing the importance of pattern recognition coupled to expressive input devices, through the use of thumb posture classification and voice input. UNC's *3DM* [5] is a three-dimensional surface modeling application using a head mounted display and a custom made 6D mouse. In both systems, the user interface takes little profit of 3D space, mostly using three-dimensional menus and limiting direct manipulation to point dragging and transformation specification. Xerox Parc's *Information Visualizer*, built using the *Cognitive Coprocessor* architecture, takes advantage of the greater possibilities of 3D with novel means of information presentation, such as the cone tree and the perspective wall, demonstrating the potential of 3D interfaces [29][8][28][23]. *MR* [34] and *Bolio* [41] are general purpose packages for building interactive 3D systems using multiple input/output devices. *MR* concentrates on the integration of devices while *Bolio* focuses on the construction of event-driven simulation systems. The object-oriented graphical toolkits *UGA* [40][9][19], from Brown University, and *Inventor* [35], from Silicon Graphics, demonstrate how the increase in correlation between manipulation and effect on controlled objects makes three-dimensional widgets more powerful and simpler to understand than their two-dimensional counterparts.

This research work reveals the potential but also the difficulty inherent in the design of three-dimensional interaction techniques. 3D interface designers are faced with systems whose structure and behavior are generally more complex than in standard 2D applications, and have to deal with a design space for interaction tools and techniques that is larger and mostly unexplored. Moreover, as stated by Myers, "the only reliable way to generate quality interfaces is to test prototypes with users and modify the design based on their comments" [26]. User interface tools, such as toolkits or frameworks, have to be used in this iterative process to reduce development time. The lack of experience in 3D interfaces makes it particularly important for these tools not to enforce any particular interface style, but to provide a wide range of interaction components, to allow rapid prototyping and testing of novel interaction techniques.

In this paper we present the *Virtuality Builder II* (*VB2*) architecture developed at the Swiss Federal Institute of Technology. The goal of *VB2* is to allow us to experiment with 3D interaction techniques and to provide a basis for the construction of our interactive applications.

2. SYSTEM STRUCTURE

VB2 is an object-oriented architecture designed to allow rapid construction of applications using a variety of 3D devices and interaction techniques. The goal of the system is to put the user in the loop of a real-time simulation, immersed in a world which can be both autonomous and dynamically responsive to its actions.

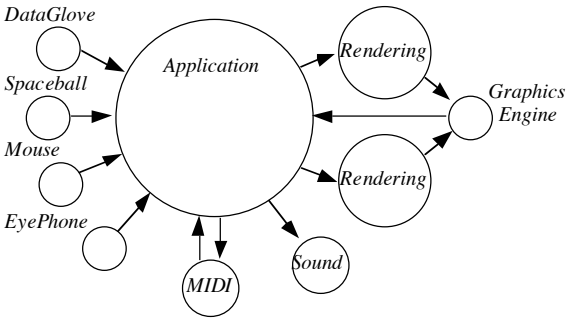


Figure 1. Overall structure of *VB2*

A *VB2* application is composed of a group of processes communicating through inter-process communication (IPC). Figure 1 shows the typical configuration of an immersive application. Processes are represented as circles, while arrows indicate the information flow between them. As in the *Decoupled Simulation Model* [34], each of the processes is continuously running, producing and consuming asynchronous messages to perform its task. A central application process manages the model of the virtual world, and simulates its evolution in response to events coming from the processes that are responsible for reading the input device sensors at specified frequencies. Sensory feedback to the user can be provided by several output devices. Visual feedback is provided by real-time rendering on graphics workstations, while audio feedback is provided by MIDI output and playback of prerecorded sounds.

The application process is by far the most complex component of the system. This process has to respond to asynchronous events by making the virtual world's model evolve from one coherent state to the next and by triggering appropriate visual and audio feedback. During interaction, the user is the source of a flow of information propagating from input device sensors to manipulated models. Multiple mediators can be interposed between sensors and models in order to transform the information accordingly to interaction metaphors.

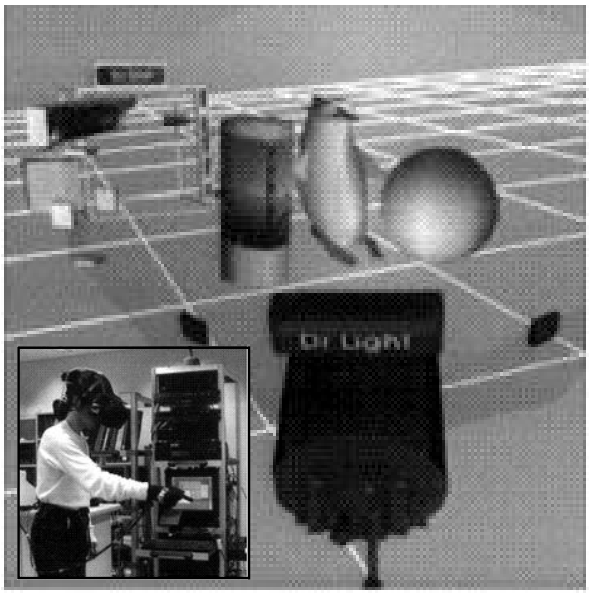


Figure 2. Synthetic environment

The remainder of the paper describes the various aspects of *VB2* application processes, with an emphasis on the dynamic model and on the interaction metaphors. First, we will give an overview of the representation of dynamic objects and of their dependencies. Next, we wi'l concentrate on how the user interacts with dynamic models through direct manipulation, gestures and virtual tools, and on how the various interaction metaphors are realized in *VB2*. The paper concludes with a discussion of the results obtained and a view of future work.

3. DYNAMIC MODEL

In order to obtain animated and interactive behavior, the system has to update its state in response to changes initiated by sensors attached to asynchronous input devices such as timers or trackers. The application can be viewed as a network of interrelated objects whose behavior is specified by the actions taken in response to changes in the objects on which they depend.

Imperative object-oriented techniques offer appropriate abstractions for representing application components and sequencing relations between states, but little support in specifying relationships between objects, since relationships are not easily encapsulated within the objects concerned [12][13][3]. The maintenance of these relationships has to be delegated to a *change propagation* mechanism responsible of updating dependent objects in response to changes in the objects on which they depend. The specification and maintenance of dependencies between objects is one of the major problems when

building reactive applications, and system performance is largely dependent on the cost of their evaluation. As an example, Conner et al. reported that part of the complexity in defining new widgets in *UGA* is due to limitations of their dependency mechanism, and a large portion of CPU time was spent in dependency evaluation [9].

Several techniques can be used to realize change propagation in object-oriented architectures, including using communication mechanisms such as *MVC* [21], distributed event handling [37][35], one-way constraints [15][40], multi-way constraints [4][22], or predefined constraints on primitive objects [1][18][20]. Constraints, being primarily declarative, free the programmer from the arduous task of maintaining relationships by hand of communication and event handling mechanisms. However, the drawback of constraint satisfaction algorithms is their limitation to specific domains or types of constraints. These two factors also determine the time complexity of such algorithms.

In order to provide a maintenance mechanism taht is both general enough to allow the specification of general dependencies between objects and efficient enough to be used in highly responsive interactive systems, we decided to model the various aspects of the system's state and behavior using different primitive elements:

- *active variables* are used to store the state of the system;
- domain-independent *hierarchical constraints*, to declaratively represent long-lived multi-way relations between active variables;
- *daemons* to react to variable changes for imperatively sequencing between different system states.

In this way, imperative and declarative programming techniques can be freely mixed to model each aspect of the system with the most appropriate means. The system's description becomes largely static, and its behavior specified by the set of active constraints and daemons. A central state manager is responsible for adding, removing, and maintaining all active constraints using an efficient local propagation algorithm, as well as managing the system time and activating daemons.

In the following sections, we will give more details on the different components and outline the state manager's behavior.

3.1 Components of the Dynamic Model

3.1.1 Active Variables and Information Modules

Active variables are the primitive elements used to store the system state. An active variable maintains its value and keeps track of its state changes. Upon request, an active variable can also maintain the history of its past values. A variable's history can be accessed using the variable's local time, which is incremented at each variable's state change, or using the system's global time. By default, global time is advanced at each constraint operation, but it is also possible to specify sequences of constraint operations to be executed within the same time slice by explicitly parenthesizing them, much as in the programming language *Kaleidoscope* [12][13]. This simple model makes it possible to elegantly express time-dependent behavior by creating constraints or daemons that refer to past values of active variables.

All *VB2* objects are instances of classes in which dynamically changing information is defined with active variables related through hierarchical constraints. Grouping active variables and constraints in classes permits the definition of *information modules* that provide levels of abstraction that can be composed to build more sophisticated behavior. Modifying some active variables of an information module is performed inside a *transaction*. Transactions are used to group changes on active variables of the same module. A module can register *reaction* objects with a set of active variables for activation at the end of transactions. Reactions are used to enforce object invariant properties as well as to maintain relationships between sets of active variables that cannot be expressed through regular constraints. A typical use of reactions is to trigger corrective actions that keep a variable's value within its limits. The reaction code is imperative and may result in the opening of new transactions on other modules as well as in the invalidation of the value of modified variables. All the operations performed during a transaction are considered as occurring within the same time slice.

3.1.2 Hierarchical Constraints

Multi-way relations between active variables are specified in *VB2* through *hierarchical constraints*, introduced in *ThingLab II* [4]. To support local propagation, constraint objects are composed of a declarative part defining the type of relation that has to be maintained and the set of constrained variables, as well as of an imperative part, the list of possible methods that could be selected by the constraint solver to maintain the constraint.

Constraint methods are not limited to simple algebraic expressions but can be general side-effect free procedures that ensure the satisfaction of the constraint after their execution by computing some of the constrained variables as a function of the others. Algorithms such as inverse geometric control of articulated chains, state machines, or non-numerical relations such as maintaining textual representations of various values, can be represented as constraint methods. This kind of generality is essential for constraints to be able to model all the various aspects of an interactive application.

A *priority level* is associated with each constraint to define the order in which constraints need to be satisfied in case of conflicts. In this way, both required and preferred constraints can be defined for the same active variable. Constraints themselves are information modules, and their priority level, as well as their boolean activation state are represented by active variables. This makes constraints full-fledged constrainable objects and allows the specification of higher-order constraints that act on other constraints to activate or deactivate them, as well as of meta-constraints that change other constraint priorities in response to the change of some variable.

3.1.3 Daemons

Daemons are the imperative portion of *VB2*. They are objects which permit the definition of sequencing between

system states. Daemons register themselves with a set of active variables and are activated each time their value changes. The action taken by a daemon can be a procedure of any complexity that may create new objects, perform input/output operations, change active variables' values, manipulate the constraint graph, or activate and deactivate other daemons. The execution of a daemon's action is sequential and each manipulation of the constraint graph advances the global system time. Daemons are executed in order of their activation time, which corresponds to breadth-first traversal of the dependency graph. Daemons can thus be used to perform discrete simulations, as it is done in *Bolio*† [41]. Examples of *VB2*'s daemons are inverse kinematics simulation for articulated chains and scene rendering triggers.

3.1.4 Variable Paths

In *VB2*, daemons, reactions, and constraints locate the variables through *indirect paths.* An indirect path is an object able to compute the location of a variable as well as the list of intermediary variables used to make its decision. Active variables are viewed in this context as self-referencing indirect paths using no intermediary variables. When a path is not capable of locating the variable, it is said to be *broken*.

A simple example of indirect paths is *symbolic paths* which correspond to Garnet's pointer variables [38]. A symbolic path is an indirect reference to a variable described by the sequence of symbolic names of the active variables that have to be traversed to reach the referenced variable. Another example is *alternative paths* which determine a variable by choosing the first successful path in a sequence.

Most of the daemons and the constraints in our system make use of indirect path definitions to locate their variables. In fact, as stated by Vander Zanden et al. [38], the use of indirect paths allows constraints to model a wide array of dynamic application behavior, and promotes a simpler, more effective style of programming than conventional constraints.

3.2 Dependency Maintenance

A central *state manager* ensures the coherent evolution of the system's state by keeping the constraint network up-to-date, triggering the execution of reactions and daemons at appropriate times, and maintaining indirect paths and variables' history. The primitive operations that the state manager performs are advancing the time, and activating or deactivating constraints, reactions, and daemons. Assigning a new value to a variable semantically corresponds to the activation immediately followed by the deactivation of an editing constraint that sets the value.

3.2.1 Activating and Deactivating Reactions and Daemons

The first operation performed to activate a reaction or a daemon is to communicate to the state manager all of its paths' intermediary variables. If none of the paths were broken, the object is registered to the variables located by its paths and is ready to react to the variables' changes. In the case of broken paths, no registration is done with variables, and the object stays dormant until all paths can be successfully resolved (see section 3.2.3). Deactivation is obtained by unregistering the object from all variables.

3.2.2 Activating and Deactivating Constraints

In order to be added to the constraint network, a constraint has to first communicate to the state manager all of its paths' intermediary variables. If one or more paths were broken, the constraint is left unenforced and the algorithm terminates. Otherwise the constraint is registered to its variables and the state manager is then asked to enforce it. Constraint deactivation is very similar to constraint activation. In this case, the constraint is first unregistered from all its variables. Then, all unenforced constraints that could potentially be enforced after the constraint deactivation are collected and the constraint manager is asked to enforce them. These constraints are the ones whose priority is less than or equal to the removed constraint's priority and that have potential output variables lying in the portion of the graph that was affected by the removed constraint.

3.2.3 Enforcing Constraints

The algorithm that attempts to enforce a set of registered unenforced constraints is the central component of the state manager. This algorithm has to find the optimal constraint graph, to update all changed variables, to handle modifications in broken paths, and to collect all the information needed to update the history of variables and to execute reactions and daemons.

The constraint solver used in *VB2* is based on the *SkyBlue* [32] local propagation algorithm, a successor of *DeltaBlue* [11] able to handle hierarchical constraints composed of methods having multiple outputs. The *SkyBlue* constraint satisfier is very efficient and domain-independent, since the algorithm consists on performing method selection on the basis of constraint priorities and graph structure alone. Furthermore, the fact that variables' values are not used by the constraint solver allows an effective application of a lazy evaluation strategy for variables. The main drawback of such local propagation algorithms is their limitation to acyclical constraint graphs. However, as noted by Maloney et al. [24], cyclical constraint networks are seldom encountered in the construction of user interfaces, and limiting the constraint solver to graphs without cycles gives enough efficiency and flexibility to create highly responsive complex interactive systems.

The complete process of updating the constraint network is described by the pseudo-code fragment of figure 3. Once *SkyBlue* has indicated which constraints must select a new method to obtain an optimal network incorporating the new constraints, the variables affected by these changes are evaluated using the method that previously determined their value and assigned to their new method. Then, all variables downstream of changes are traversed to mark them out-of-date and to collect all dependent objects. At the end of the propagation, all

† *VB2*'s daemons correspond to *Bolio*'s "constraint modules".

objects that used a now modified variable to compute indirect deactivated and reactivated to reconnect them to the correct variables, as in the user-interface toolkit *Multi-Garnet* [33], and all transactions opened during propagation are closed to execute the reactions. In practice, the planning phase (corresponding to method selection) and the propagation phase can be separated, so as to cache constraint plans and to reuse them when still valid (for example when repeatedly assigning to a variable that is only used as input in a stable constraint network).

```
let T be the set of objects on which the state manager opens transactions
let H be the set of variables needing history update
let B be the set of objects with broken paths
let D be the queue of pending daemons
Find an optimal constraint graph:
    Use SkyBlue to select new computing methods for a set of constraints
Update the method graph:
    for each variable v that will change computing method do
        if not v.is_evaluated then
            v.computing_method.execute
    for each variable v that has to change computing method do
        v.computing_method:= computing method determined by SkyBlue
Propagate the changes:
    for each variable v downstream of constraints with a new method do
        if v.owner is not inside a transaction then
            v.owner.open_transaction
            add v.owner to T
        v.owner.collect_reactions(v)
        v.is_evaluated:= False
        v.time:= current_time
        if v keeps track of history then
            add v to H
        append v.daemons to D
        add all objects that used v for their paths to B
    while B not empty do
        remove o from B
        o.deactivate
        o.activate
    while T not empty do
        remove o from T
        o.close_transaction
```

Figure 3. Constraint graph update

3.2.4 Advancing the Time

At the end of a time slice, all variables needing a history update are evaluated and their value is stored in their history list. Once done, the time is advanced, and the pending daemons are extracted from the queue and executed one after the other in the order of their priority. This process continues until the queue becomes empty. Each daemon activation may result in series of recursive calls to the state manager caused by constraint operations, propagation of values, or time increments.

3.3 Defining Complex Dynamic Objects

All the dynamic attributes of *VB2*'s classes are represented with active variables while their behavior is defined by an internal constraint network. Active variables store the objects' state, while internal constraints implement the objects' behavior. Internal integrity constraints have maximum priority to ensure that objects' invariant properties are always satisfied. For example, we used this approach for the design of *VB2*'s modeling class cluster, whose basic structure is presented in figure 5. Figure 4 shows the design notation used.

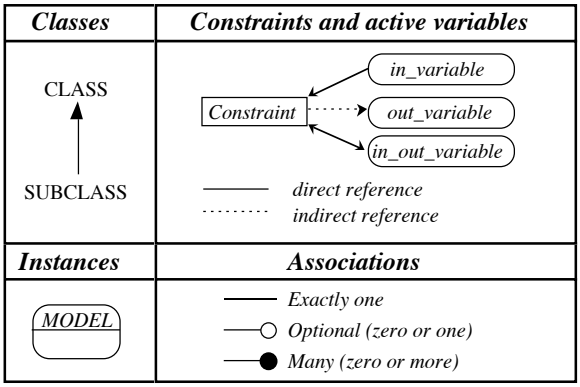


Figure 4. Design notation

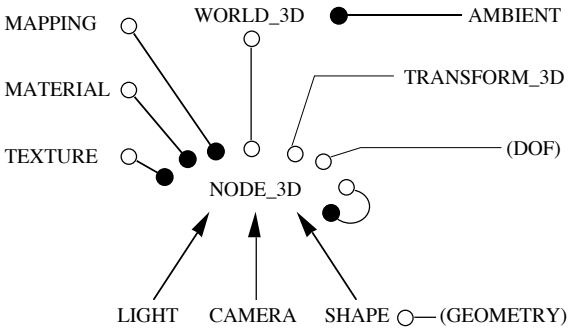


Figure 5. Basic modeling class hierarchy.

The central component of this cluster is the NODE_3D class, whose instances, related in a hierarchical fashion, represent the transformation hierarchy. Position, orientation, shearing and scaling of the reference frame are packaged in TRANSFORM_3D objects. Degrees of freedom can be attached to a node in order to define additional constrained motion, as in articulated structures. Instances of MATERIAL and TEXTURE are used to define the behavior of physical objects with respect to light. Placing instances of MATERIAL and TEXTURE in a node allows instance inheritance through the hierarchy. Instances of LIGHT represents light sources whose color and intensity is defined by instances of MATERIAL and TEXTURE. An instance of CAMERA represents a camera viewing the scene. It maintains information about its viewing frustrum and a possibly stereoscopic projection. Instances of SHAPE encapsulate the concept of physical objects having a geometry, material and texture in the Cartesian space. More details on the class hierarchy of the modeling and rendering clusters are presented in [17].

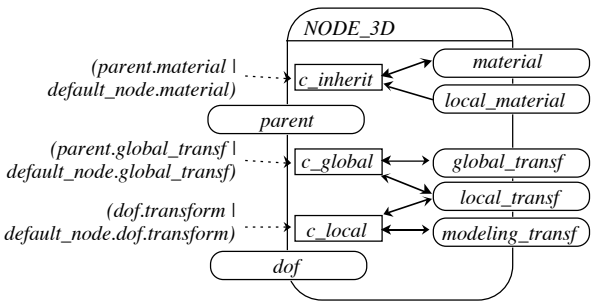


Figure 6. NODE_3D's simplified constraint network.

The use of indirect paths allows the declarative specification of structured objects. Figure 6 illustrates the use of symbolic and alternative paths in the definition of NODE_3D's internal constraint network. The constraints *c_local* and *c_global* maintains all transformations up-to-date, and the constraint *c_inherit* realizes attribute inheritance through the instance hierarchy.

4. DYNAMICS AND INTERACTION

Animated and interactive behavior can be thought of together as the fundamental problem of dynamic graphics: how to modify graphical output in response to input? Time-varying behavior is obtained by mapping dynamically changing values, representing data coming from input devices or animation scripts, to variables in the virtual world's model. The definition of this mapping is crucial for interactive applications, because it defines the way users communicate with the computer. Ideally interactive 3D systems should allow users to interact with synthetic worlds in the same way they interact with the real world, thus making the interaction task more natural and reducing training.

4.1 Mapping Sensor Measurements to Actions

In most typical interactive applications, users spend a large part of their time entering information, and several types of input devices, such as 3D mice and DataGloves, are used to let them interact with the virtual world. Using these devices, the user has to provide at high speed a complex flow of information, and a mapping has to be devised between the information coming from the sensors attached to the devices and the actions in the virtual world. Most of the time, this mapping is hard coded and directly dependent on the physical structure of the device used (for example, by associating different actions to the various mouse buttons). This kind of behavior is obtained in *VB2* by attaching constraints directly relating the sensors' active variables to variables in the dynamic model, as in the example of figure 7. The beginning of the direct manipulation of a model is determined by the activation of a constraint between input sensor variables and some of the active variables in the interface of the model. While the interaction constraint remains active, the user can manipulate the model through the provided metaphor. The deactivation of the interaction constraint terminates the direct manipulation. Second-order constraints that depend on boolean state variables are generally used to trigger activation and deactivation of interaction constraints.

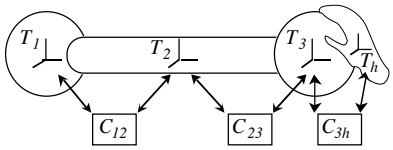


Figure 7. Graphical objects grabbed by user with constraints

Such a direct mapping between the device and the dynamic model is straightforward to choose for tasks where the relations between the user's motions and the desired effect in the virtual world is mostly physical, as in the example of grabbing an object and moving it, but needs to be very carefully thought out for tasks where user's motions are intended to carry out a meaning. In this latter case, hardwiring virtual world actions to specific sensor values forces commitments that would risk reducing device expressiveness and can make applications difficult to use [10].

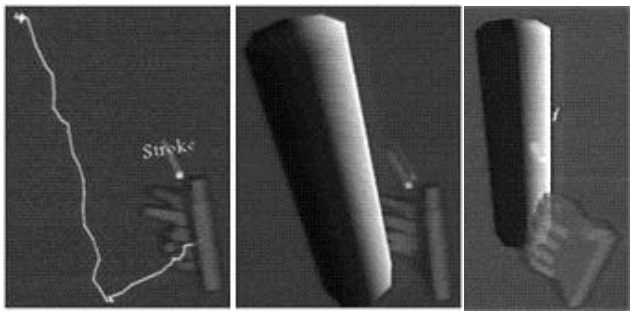
Adaptive pattern recognition can be used to overcome these problems, by letting the definition of the mapping between sensor measurements and actions in the virtual world be more complex, and therefore increasing the expressive power of the devices. Furthermore, the possibility of specifying this mapping through examples makes applications easier to adapt to the preferences of new users, and thus simpler to use.

4.1.1 Hand Gestures

VB2 uses a gesture recognition system linked to the *DataGlove*. Whole-hand input is emerging as a research topic in itself, and some sort of posture or gesture recognition is now being used in many virtual reality systems (see Sturman [36] for a detailed overview of whole-hand input). The gesture recognition system has to classify movements and configurations of the hand in different categories on the basis of previously seen examples. Once the gesture is classified, parametric information for that gesture can be extracted from the way it was performed, and an action in the virtual world can be executed. In this way, with a single gesture both categorical and parametric information can be provided at the same time in a natural way [30]. A visual and an audio feedback on the type of gesture recognized and on the actions executed are usually provided in *VB2* applications to help the user understand system's behavior.

VB2's gesture recognition is subdivided into two main portions: posture recognition, and path recognition. The posture recognition subsystem is continuously running and is responsible for classifying the user's finger configurations. Once a configuration has been recognized, the hand data is accumulated as long as the hand remains in the same posture. The history mechanism of active variables is used to automatically perform this accumulation. This data is then passed to the path recognition subsystem to classify the path. A gesture is therefore defined as the path of the hand while the hand fingers remain stable in a recognized posture. The type of gesture chosen is compatible with Buxton's suggestion [6][7] of using physical tension as a natural criterion for segmenting primitive interactions: the user, starting from a relaxed state, begins a primitive interaction by tensing some muscles and raising its state of attentiveness, performs the interaction, and then relaxes the muscles. In our case, the beginning of an interaction is indicated by positioning the hand in a recognizable posture, and the end of the interaction by relaxing the fingers. One of the main advantages of this technique is that, since postures are static, the learning process can be done interactively by putting the hand in the right position and indicating when to sample to the computer. Once postures are learnt, the paths can be similarly learnt in an interactive way, using the posture classifier to correctly segment the input when

generating the examples. Many types of classifiers could be used for the learning and recognition task. In the current implementation of *VB2*, feature vectors are extracted from the raw sensor data, and *multi-layer perceptron networks* [31] are used to approximate the functions that map these vectors to their respective classes [16].



(a) (b) (c)

Figure 8a, 8b. Creating a cylinder by gestural input
Figure 8c. Grabbing the cylinder through posture recognition

The gesture recognition system is a way to enhance the data coming from the sensors with classification information and thus provides an *augmented interface* to the device. This is modeled in *VB2* by explicitly representing these higher-level views of devices as dynamic objects with a set of active variables representing the augmented information, the gesture-recognition system being represented as a multiple-output constraint responsible for maintaining the consistency between the device data and the high-level view. Application objects can then bind constraints and daemons to both low- and high-level active variables to program their behavior. Figure 9 shows how this is realized for the *DataGlove*. Other more abstract views of devices may be provided by adding other constraint networks linking abstract device objects to more device-dependent views.

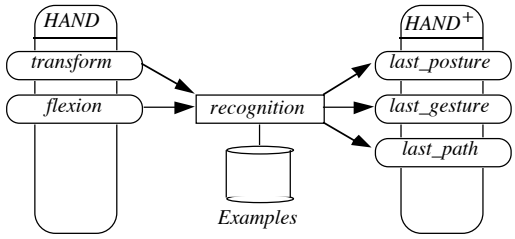


Figure 9. DataGlove device object and gestural interface

4.2 Virtual Tools

The amount of information that can be controlled on a three-dimensional object and the ways that could be used to control it are enormous. Gestural input techniques and direct manipulation on the objects themselves offer only partial solutions to the interaction problem, because these techniques imply that the user knows what can be manipulated on an object and how to do it. The system can guide the user to understand a model's behavior and interaction metaphors by using mediator objects that present a selective view of the model's information and offer the interaction metaphor to control this information. We call these objects *virtual tools*.

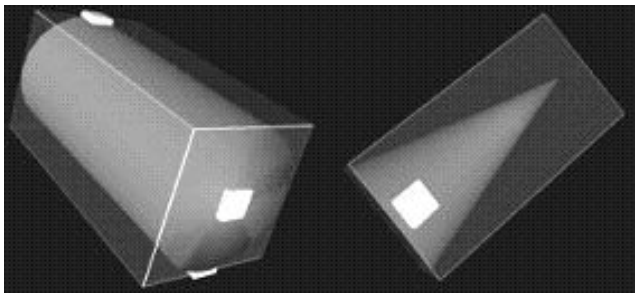


Figure 10. Examples of simple virtual tools

VB2's virtual tools are first class objects, like the widgets of *UGA* [9], which encapsulate a visual appearance and a behavior to control and display information about application objects. The visual appearance of a tool must provide information about its behavior and offer visual semantic feedback to the user during manipulation.

Designing interaction tools is a difficult task, especially in 3D where the number of degrees of freedom is much larger than in 2D. Therefore, experimentation is necessary to determine which tools are needed and how these tools must be organized to build a powerful workspace. In *VB2*, virtual tools are fully part of the synthetic environment. As in the real world, the user configures its workspace by selecting tools, positioning and orienting them in space, and binding them to the models he intends to manipulate. When the user binds a tool to a model, he initiates a bi-directional information communication between these two objects which conforms with the multiple-threaded style of man-machine dialogue supported by *VB2*. Multiple tools may be attached to a single model in order to simultaneously manipulate different parts of the model's information, or the same parts using multiple interaction metaphors.

The tool's behavior must ensure the consistency between its visual appearance and the information about the model being manipulated, as well as allow information editing through a physical metaphor. In *VB2*, the tool's behavior is defined as an internal constraint network, while the information required to perform the manipulation is represented by a set of active variables. The models that can be manipulated by a tool are those whose external interface matches that of the tool. The visual appearance is described using a modeling hierarchy. In fact, most of our tools are defined as articulated structures that can be manipulated using inverse kinematics techniques, as tools can often be associated with mechanical systems.

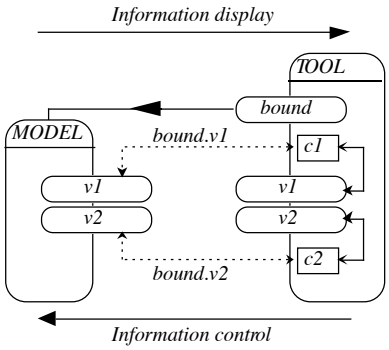


Figure 11. Model and virtual tool

4.2.1 Virtual Tool Protocol

The user declares the desire to manipulate an object with a tool by *binding* a model to a tool. When a tool is bound, the user can manipulate the model using it, until he decides to *unbind* it.

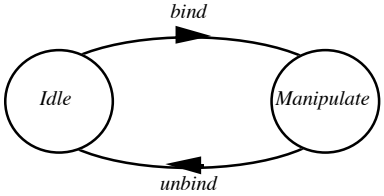


Figure 12. Tool's state transitions

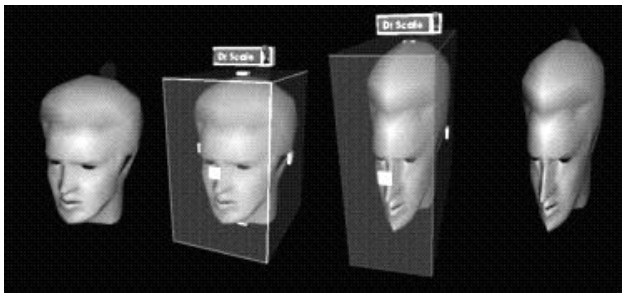
Tools have a *bound* active variable that references the manipulated model. Binding a model to a tool consists of assigning to *bound* a reference to a manipulatable model, while setting *bound* to a void reference will unbind the tool.

When *binding* a model to a tool, the tool must first determine if it can manipulate the given model, identifying on the model the set of public active variables requested to activate its binding constraints. Once the binding constraints are activated, the model is ready to be manipulated. The binding constraints being generally bi-directional, the tool is always forced to reflect the information present in the model even if it is modified by other objects.

When a tool is bound to a model, the user can manipulate the model's information through a physical metaphor. This iterative process composed of elementary manipulations is started by the selection of some part of the tool by the user, resulting in the activation of some constraint such as, for example, a motion control constraint between the 3D cursor and the selected part. User input motion results in changes to the model's information by propagation of device sensor values through the tool's constraint network, until the user completes the manipulation by deselecting the tool's part. Gestural input techniques can be used to initiate and control a tool's manipulations, for example by associating selection and deselection operations to specific hand postures.

Unbinding a model from a tool detaches it from the object it controls. The effect is to deactivate the binding constraints in order to suppress dependencies between tool's and model's active variables. Once the model is unbound, further manipulation of the tool will have no effect on the model.

All binding constraints reference the model's variables using indirect paths through the tool's *bound* variable. Second-order control is used to ensure simultaneous activation and deactivation of all the tool's binding constraints every time the value of the *bound* variable changes.



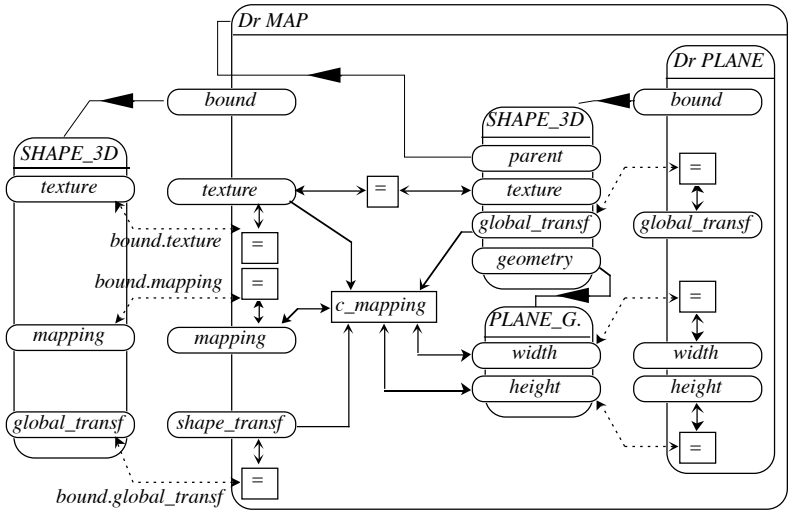
(a). (b) (c) (d)

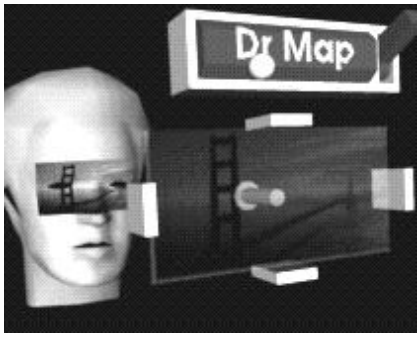
Figure 13a. Model before manipulation
Figure 13b. A scale tool is made visible and bound to the model
Figure 13c. The model is manipulated via the scale tool
Figure 13d. The scale tool is unbound and made invisible

4.2.2 A Simple Tool: Dr. Plane

Dr. Plane is a tool that manipulates a shape whose geometry is a plane. In *VB2*, a plane geometry is a meshed object defined on the plane *XY* and defined by two active variables, its width and its height. The information required by the tool to achieve manipulation is composed of three variables: the width and height of the plane, used to control its size, and its global transformation, used to ensure that the tool's position and orientation reflect those of the manipulated shape. The visual appearance of the tool is defined as a set of four markers, two for the display and manipulation of the width information and two for the height. This redundancy is introduced so that one of the markers be always accessible from any viewpoint. Each marker is associated with a single translational degree of freedom between the origin and the border of the plane. Width control and display is achieved by placing equality constraints between the value of the two degrees of freedom associated with the width markers. The width variable is constrained to be equal the value of one of the degrees of freedom. Height manipulation is implemented similarly.

When binding a model to *Dr. Plane*, three equality constraints are started between the width variables, the height variables, and the global transformation variables of the shape and tool. Once the binding constraints are activated, selection of a marker will start an inverse kinematics constraint between the cursor and the marker, which will modify the value of the degree of freedom and therefore the width or height of the plane according to user's motion. Similarly, modification of the plane's parameters by program or by another tool will result into a





(a) (b)

Figure 15a. *Dr. Map*'s simplified constraint network
Figure 15b. View of *Dr. Map*

motion of the degrees of freedom positioning the markers at the correct new position. The constraint between the global transformation variables ensures that the tool will always be placed around the model it manipulates, even if the model is manipulated by program or by other tools.

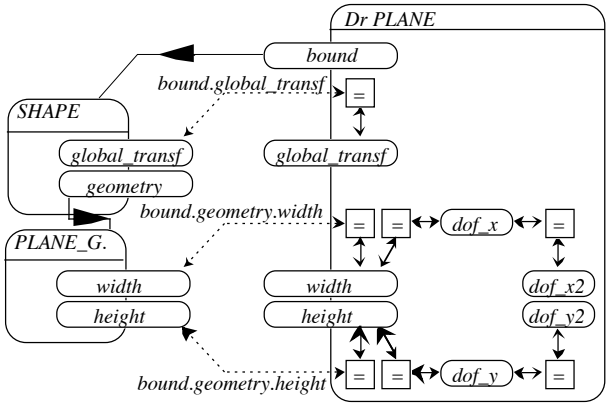


Figure 14a. *Dr. Plane*'s simplified constraint network

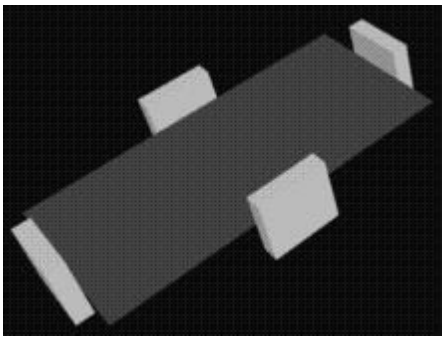


Figure 14b. View of *Dr. Plane*

4.2.3 Composition of Virtual Tools

Since virtual tools are first class dynamic objects in *VB2*, they can be assembled into more complex tools much in the same way simple tools are built on top of a modeling hierarchy. The reuse of abstractions provided by this solution is far more important than the more obvious reuse of code.

An example of a composite tool is *Dr. Map*, which is a virtual tool used to edit the texture mapping function of a model by controlling the parallel projection of an image on the surface of the manipulated model. The tool is defined as a plane on top of which is mapped the texture, a small arrow icon displaying the direction of projection. In order to compute the mapping function to be applied to the model, the tool needs to know the texture to be used, the position and orientation of the model in space, and the position and orientation of the tool in space. The textured plane represents the image being mapped, and a *Dr. Plane* tool allows manipulation of the plane in order to change the aspect ratio of the texture's image. The constraint *c_mapping* uses the model's and tool's transformations, the texture, and the width and height values to maintain the mapping function.

Similarly, the material editing tool is built out of a color tool and the light tool is built out of a cone tool. By reusing other tools we enforce consistency of the interface over the entire system, allowing users to perceive rapidly the actions they can perform. Building tools by composing the behavior and appearance of simpler objects is relatively easy in *VB2*: for example, *Dr. Map* tool was built and tested by one person in less than a couple of hours. The fast prototyping capabilities of the system are very important for an architecture aimed at experimenting with 3D interaction.

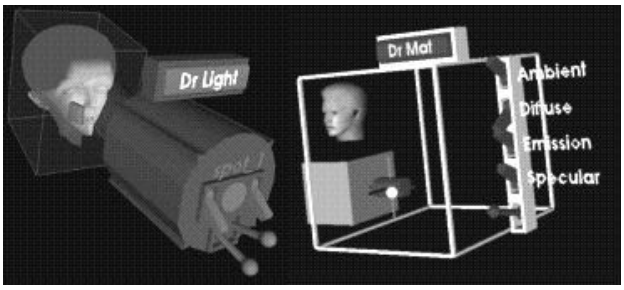


Figure 16. View of some other composite tools

5. IMPLEMENTATION AND RESULTS

VB2 is implemented in the object-oriented language *Eiffel* [25] on *Silicon Graphics* workstations, and is currently composed of over 500 classes. The 3D input devices currently available are the *Spaceball* and the *VPL DataGlove*. Audio feedback is provided through MIDI output or by triggering playback of prerecorded sounds on a *NeXT Cube*. Visual feedback is provided by rendering a pair of stereo images on one graphics workstation connected to a *VPL Eyephone* through a custom-made image splitter. In the current implementation of *VB2*, rendering is performed directly by the application process.

Complex applications composed of thousands of variables and constraints can be run at interactive speed. The performance analysis of full scale 3D applications shows that the redraw speed of the hardware is the limiting factor on interaction speed. Figure 17 presents the constraint network statistics of the immersive application of figure 2 at the moment corresponding to that frame. The number of triangles composing the scene was 5905. The frame rate of the application was 8 Hz, and the average redraw time was 110 ms, leaving only 15 ms to perform the other system's tasks.

Constraints	:	*1677*	
Active constraints	:	*1521*	*(91 % of constraints)*
Indirect constraints	:	*1120*	*(74 % of active constraints)*
Variables	:	*3514*	
Free variables	:	*1110*	*(32 % of variables)*
Constrained variables	:	*2404*	*(68 % of variables)*
Pure input variables	:	*884*	*(37 % of constrained variables)*
Pure output variables	:	*445*	*(19 % of constrained variables)*
Input/output variables	:	*1075*	*(45 % of constrained variables)*
Unevaluated variables	:	*557*	*(37 % of out and in/out variables)*

Figure 17. Constraint network statistics of an immersive application

All the statistics presented in this section were obtained on a *Silicon Graphics Crimson VGX*.

5. CONCLUSIONS AND FURTHER WORK

We have presented the *VB2* architecture for the construction of three-dimensional interactive applications. A *VB2* application is composed of a group of continuously running processes that asynchronously produce and consume IPC messages to perform their tasks. A central application process manages the virtual world's model and simulates its evolution in response to events coming from the processes that encapsulate the input devices. Multiple devices can be used to interact with the synthetic world through various interaction paradigms. Interaction techniques range from direct manipulation, to gestural input and three-dimensional virtual tools. Adaptive pattern recognition is used to increase input device expressiveness by enhancing sensor data with classification information. Tools, encapsulations of visual appearance and behavior, present a selective view of the manipulated model's information and offer the interaction metaphor to control it. Since tools are first class objects, they can be assembled into more complex tools, much in the same way simple tools are built on top of a modeling hierarchy. New three-dimensional tools are easily added to the system, and their number is rapidly growing.

Hierarchical constraints, active variables, and daemons are used to uniformly represent the system state and behavior. The use of an incremental constraint solver based on an enhancement of *SkyBlue* makes it possible to run, at interactive speed, complex applications composed of thousands of variables and constraints. The redraw time of the hardware is still the limiting factor on interaction speed.

We believe that *VB2* provides a good platform for prototyping and integrating a large variety of three-dimensional interaction metaphors to control all the different aspects of synthetic environments. We are currently extending the architecture with tools for animation control in order to build a virtual reality animation system.

ACKNOWLEDGMENTS

We would like to thank Michel Gangnet, Geoff Wyvill for reviewing this paper, Gilles Van Ruymbeke and Marc Ledin for building the *VB2* image splitter, and Angelo Mangili and Russell Turner for participation in the design and implementation of an early version of *VB2*.

This research was partly sponsored by *Le Fonds National Suisse pour la Recherche Scientifique*.

REFERENCES

[1] Avesani P, Perini A, Ricci F (1990), COOL: An Object System with Constraints. *Proc. TOOLS 2*.

[2] Balaguer JF, Mangili A (1992), Virtual Environments. In Thalmann D, Magnenat-Thalmann N (Editors) *New Trends in Animation and Visualization*, John Wiley and Sons: 91-105.

[3] Blake E, Hoole Q (1992), Expressing Relationships Between Objects: Problems and Solutions. *Proc. Third EUROGRAPHICS Workshop on Object-Oriented Graphics:* 159-162.

[4] Borning A, Duisberg R, Freeman-Benson B, Kramer A, Woolf M (1987), Constraint Hierarchies, *Proc. OOPSLA*: 48-60.

[5] Butterworth J., Davidson A., Hench S., Olano TM (1992), 3DM: A Three Dimensional Modeler Using a Head-Mounted Display. *Proc. SIGGRAPH Symposium on Interactive 3D Graphics*: 135-138.

[6] Buxton WAS (1986), Chunking and Phrasing and the Design of Human-Computer Dialogues. In *Information Processing*. North Holland. Elsevier Science Publishers.

[7] Buxton WAS (1990), A Three-state model of Graphical Input. In Diaper D, Gilmore D, Cockton G, Shackel B (Editors) *Human-Computer Interaction: Interact, Proceedings of the IFIP Third International Conference on Human-Computer Interaction,* North-Holland, Oxford.

[8] Card SK, Robertson GG, Mackinlay JD (1991), The Information Visualizer, An Information Workspace. *Proc. SIGCHI*: 181-188.

[9] Conner DB, Snibbe SS, Herndon KP, Robbins DC, Zeleznik RC, Van Dam A (1992), Three-Dimensional Widgets. *SIGGRAPH Symposium on Interactive 3D Graphics*: 183-188.

[10] Fels SS, Hinton GE (1990), Building Adaptive Interfaces with Neural Networks: The Glove-Talk Pilot Study. In Diaper D, Gilmore D, Cockton G, Shackel B (Editors) *Human-Computer Interaction: Interact, Proceedings of the IFIP Third International Conference on Human-Computer Interaction*, North-Holland, Oxford: 683-687.

[11] Freeman-Benson BM, Maloney A (1989), The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. In *Proceedings of the Eighth Annual IEEE International Phoenix Conference on Computers and Communications,* March.

[12] Freeman-Benson BN (1990), Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. *Proc. ECOOP/OOPSLA:* 77-87.

[13] Freeman-Benson BN, Borning A (1992), Integrating Constraints with an Object-Oriented Language. *Proc. ECOOP:* 268-286.

[14] Gleicher M, Witkin A (1991), Snap Together Mathematics. In Blake EH, Wisskirchen P (Editors) *Advances in Object-Oriented Graphics 1: Proc. EUROGRAPHICS Workshop on Object-Oriented Graphics*: 21-34.

[15] Giuse D (1992), KR: Constraint-Based Knowledge Representation, Technical Report, Carnegie-Mellon University.

[16] Gobbetti E (1992), *Reconnaissance de gestes pour l'interaction*, Technical Report, EPFL/DI-LIG.

[17] Gobbetti E, Balaguer JF, Mangili A, Turner R (1993), Building an Interactive 3D Animation System. In Meyer B, Nerson JM (Editors) *Object-Oriented Applications*, Prentice-Hall.

[18] Helm R, Huyhn T, Marriot K, Vlissides J (1992), An Object-Oriented Architecture for Constraint-Based Graphical Editing. *Proc. Third EUROGRAPHICS Workshop on Object-Oriented Graphics:* 1-22.

[19] Herndon KP, Zeleznik RC, Robbins DC, Conner DB, Snibbe SS and van Dam A (1992), Interactive Shadows, *Proc. UIST:* 1-6.

[20] Kass M (1992), CONDOR: Constraint-Based Dataflow. *Proc. SIGGRAPH*: 321-330.

[21] Krasner GE, Pope ST (1988), A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1(3): 26-49.

[22] Leler WM (1988), *Constraint Programming Languages: Their Specification and Generation*. Addison Wesley.

[23] Mackinlay JD, Robertson GG, Card SK (1991), The Perspective Wall: Detail and Context Smoothly Integrated. *Proc. SIGCHI*: 173-179.

[24] Maloney J, Borning A, Freeman-Benson BN (1989), Constraint Technology for User Interface Construction in ThingLab II. *Proc. OOPSLA*: 381-396.

[25] Meyer B (1992), *Eiffel: The Language*. Prentice-Hall.

[26] Myers BA (1989), User-Interface Tools: Introduction and Survey. *IEEE Software*. 6(1): 15-23.

[27] NSF (1992), Research Directions in Virtual Environments, *NSF Invitational Workshop,* UNC at Chapel Hill, March 23-24: 154-177.

[28] Robertson GG, Card KS, Mackinlay JD (1989), The Cognitive Coprocessor Architecture for Interactive User Interfaces. *Proc. UIST:* 10-18.

[29] Robertson GG, Mackinlay JD, Card SK (1991) Cone Trees: Animated 3D Visualizations of Hierarchical Information. *Proc. SIGCHI*: 189-194.

[30] Rubine DH (1991), *The Automatic Recognition of Gestures,* PhD Thesis, CMU-CS-91-292, Carnegie Mellon University.

[31] Rumelhart DE, Hinton GE, Williams RJ (1986), Learning Internal Representations by Error Propagation. In Rumelhart DE, McClelland JL

(Editors) *Parallel Distributed Processing,* Vol. 1: 318-362.

[32] Sannella M (1993), *The SkyBlue Constraint Solver*. TR-92-07-02, Dept. of Computer Science, University of Washington.

[33] Sannella M, Borning A (1992), *Multi-Garnet: Integrating Multi-way Constraints with Garnet*. TR-92-07-01, Dept. of Computer Science, University of Washington.

[34] Shaw C, Liang J, Green M, Sun Y (1992), The Decoupled Simulation Model for Virtual Reality Systems. *Proc. SIGCHI:* 321-328.

[35] Strauss PS, Carey R (1992), An Object-Oriented 3D Graphics Toolkit. Proc. SIGGRAPH: 341-347.

[36] Sturman DJ (1991), *Whole-Hand Input*, PhD Thesis, MIT.

[37] Turner R, Gobbetti E., Balaguer JF, Mangili A, Thalmann D, Magnenat-Thalmann N (1990), An Object-oriented Methodology with Dynamic Variables for Animation and Scientific Visualization. *Proc. CGI:* 317-328.

[38] Vander Zanden B, Myers BA, Giuse D, Szeleky P (1991), The Importance of Pointer Variables in Constraint Models. *Proc. UIST:* 155-164.

[39] Weiner D, Ganapathy SK (1989), A Synthetic Visual Environment with Hand Gesturing and Voice Input. *Proc. SIGCHI*: 235-240.

[40] Zeleznik RC, Conner DB, Wlocka MM, Aliaga DG, Wang NT, Hubbard PM, Knepp B, Kaufman H, Hughes JF, van Dam A (1991), An Object-Oriented Framework for the Integration of Interactive Animation Techniques. *Proc. SIGGRAPH*: 105-112.

[41] Zeltzer D, Pieper S, Sturman DJ (1989), An Integrated Graphical Simulation Platform, *Proc. Graphics Interface:* 266-274.